BERT for Humanists

White Paper

Digital Humanities Advancement Grant

National Endowment for the Humanities

HAA-277275-21

1/1/2021 – 12/31/2021
# Table of Contents

<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>Project Summary</td>
<td>3</td>
</tr>
<tr>
<td>Project Origins and Goals</td>
<td>3</td>
</tr>
<tr>
<td>Project Activities, Team, &amp; Participants</td>
<td>5</td>
</tr>
<tr>
<td>Project Team</td>
<td>5</td>
</tr>
<tr>
<td>Advisory Team</td>
<td>5</td>
</tr>
<tr>
<td>Project Activities</td>
<td>6</td>
</tr>
<tr>
<td>Project Outcomes</td>
<td>8</td>
</tr>
<tr>
<td>Infrastructure</td>
<td>8</td>
</tr>
<tr>
<td>Machine learning concepts</td>
<td>9</td>
</tr>
<tr>
<td>BERT-specific concepts</td>
<td>10</td>
</tr>
<tr>
<td>Use Cases</td>
<td>11</td>
</tr>
<tr>
<td>Multilingual modeling</td>
<td>14</td>
</tr>
<tr>
<td>Project Evaluation and Impact</td>
<td>14</td>
</tr>
<tr>
<td>Project Continuation and Long-Term Impact</td>
<td>15</td>
</tr>
<tr>
<td>Works Cited</td>
<td>16</td>
</tr>
<tr>
<td>Appendix</td>
<td>17</td>
</tr>
</tbody>
</table>
Project Summary

Large language models (LLMs) have revolutionized the field of natural language processing (NLP), but they raise significant challenges for researchers working with text in the humanities, especially those who do not have prior machine learning experience. To address this disconnect, our project — BERT for Humanists — developed a public-facing workshop and a suite of freely available educational materials to help digital humanities scholars learn if LLMs like BERT are appropriate for their research and, if so, how to responsibly use and understand them.

Project Origins and Goals

Humans are able to separate the general ability to speak a language, say English or Korean, from the ability to perform specific tasks using that language, say buying a muffin or writing an NEH white paper. But until recently, researchers in natural language processing (NLP) approached each new task by building an entirely new system from scratch. The innovation of large language models (LLMs) is that we can create a single general-purpose language model that can be quickly adapted to achieve strong performance on a variety of tasks (not unlike a human who can seamlessly order a muffin and then write an NEH white paper). The goal of the BERT for Humanists project is to connect this new paradigm of NLP to the needs and abilities of humanists.

The broad mission of our research group is to build methodologies that help people learn about culture through documents. We have been successful at building tools and creating procedures for scholars in humanities disciplines to use methods such as topic modeling and word embeddings in powerful and meaningful ways. This work has included building and supporting tools such as Mallet and jsLDA, as well as studying the effect of different protocols. For example, we determined that for English documents, stemming has little to no benefit in topic modeling (Schofield and Mimno 2016); and that using approximately 20 bootstrap resamplings of a collection to train multiple word embedding models provides much more stable and meaningful word similarity measurements (Antoniak and Mimno 2018).
This experience has given us a strong sense of what works in translating between NLP research and humanist practice. But it has also given us countless examples of what can go wrong. Researchers can waste valuable time using the wrong tools or asking the wrong questions. Seemingly inconsequential choices in preprocessing or analysis of results can lead to invalid or meaningless conclusions. And this doesn’t even take into account the unknown number of researchers who gave up after not being able to figure out how to apply a new tool, or who never recognized that a method could be useful in the first place.

With this context in mind, we were motivated to address how humanities researchers might use large language models. In the late 2010s, LLMs like ELMo (Peters et al. 2018) and BERT (Devlin et al. 2019) took the NLP world by storm, providing noticeably better results in a wide range of tasks and transforming the field seemingly overnight. These LLMs leverage large collections of text to generate a “pre-trained”, general-purpose language model that can be applied to a wide range of tasks as well as “fine-tuned” to specific datasets and domains — all huge advances compared to previous methods. However, in doing so, these models also bring with them a host of mathematical, technical, and infrastructural challenges that humanities researchers often do not have sufficient training or experience to overcome. So we knew two things: first, it was only a matter of time before the LLM train reached “Humanities-land”; and second, nobody could climb aboard unless the DH community built a platform to do so.

We decided to begin the work of building this platform, and we started by assessing the pain points that humanities researchers experience when using LLMs. We held an initial brainstorming/focus group session with our Advisory Team, which consists of digital humanities and natural language processing researchers who have experience using text processing methods. During the session, we discussed whether or not participants had used LLMs like BERT in the past — if so, in which settings, and if not, what obstacles had prevented them.

Based on these conversations, we then designed an interactive tutorial for digital humanists to build familiarity with LLMs. Many already existing LLM tutorials focused on the how of these models, and not enough on the why. They went straight for low-level technical details without talking about motivation or use cases. For our tutorials, we situated LLMs within a wider set of models, including
the more familiar static word embeddings, and we broadened our conversation to include non-standard use cases for LLMs that make more sense in humanities contexts.

In our technical explanations of transformer-based LLMs and the pretraining/fine-tuning framework, we prioritized high level intuitions over mathematical details. Our goals were to empower participants to make informed decisions when curating datasets and setting fine-tuning parameters, and to give participants the vocabulary and access points to learn more after the tutorial. Additionally, previous experience has shown us that machine learning tools are most useful when accompanied by humanities-focused tutorials, easy-to-use software, lots of advice about parameter setting, and protocols that are tuned to specific use cases and data patterns. With this in mind, we designed our tutorials with interactive Google Colab notebooks that required no initial setup or hardware.

Project Activities, Team, & Participants

Project Team

- Project Director, David Mimno, Associate Professor, Information Science, Cornell University
- Co-Project Director, Melanie Walsh, Postdoctoral Associate, Information Science, Cornell University
- Lead Developer, Maria Antoniak, PhD, Information Science, Cornell University

Advisory Team

Our Advisory Team consisted of scholars engaged in research at the boundary of digital humanities and natural language processing, including:

- Ted Underwood, Professor, School of Information Sciences and English, University of Illinois, Urbana-Champaign
- David Bamman, Assistant Professor, School of Information, UC Berkeley
- Lucy Li, PhD student, School of Information, UC Berkeley
- Matt Sims, Postdoctoral Scholar, School of Information, UC Berkeley
Rachel Buurma, Associate Professor, English, Swarthmore College
Alvin C. Grissom II, Assistant Professor, Computer Science, Haverford College
Peter Bol, Professor, East Asian Languages and Civilizations, Harvard University
Liu Zhou, Peking University
Hongsu Henry Wang, Research Fellow, Institute for Quantitative Social Science, Harvard University
Richard Wicentowski, Professor, Computer Science, Swarthmore College
Lauren Klein, Associate Professor, English and Quantitative Theory and Methods, Emory University
Ryan Dubnicek, Digital Humanities Specialist, HathiTrust Research Center, University of Illinois

Project Activities

Over the course of the grant period, we hosted two introductory BERT workshops, the first by invitation to a panel of digital humanities/natural language experts (March 2021) and the second to the public (June 2021). Two hundred researchers from around the world registered to participate in our public virtual workshop, and more than 90 actually attended the event. For these workshops, we developed a slide presentation and a set of humanities use cases for BERT with code-based tutorials, focusing on collections of poetry and online book reviews. To demonstrate how users might apply BERT to languages beyond English, we also included a tutorial for studying Spanish-language sonnets from the early 16th century to late 17th century. In addition to these materials, we created an annotated bibliography of BERT resources and a glossary of key terms, both of which are published on our project website. Most broadly, we solidified connections among the members of our advisory board and started to build a global, interdisciplinary community of DH/NLP researchers who are interested in LLMs.

It is important to note that there were communities we were not able to serve with our project — namely, scholars who lack programming and machine learning experience. If we prepared for participants with less experience, we could probably create a more accessible workshop, but we would do so at the cost of reducing our ability to cover BERT-specific material and of potentially losing more experienced
participants. Ultimately, we decided to assume a base level of knowledge of machine learning concepts, such as training/testing splits and binary classification.

Further details about our project activities are included in the timeline below.

**Brainstorming Session 1 — January 19, 2021**

We convened our DH-NLP Advisory Team via Zoom video conference. We discussed and took notes on how members of the advisory team have incorporated BERT into their research so far, what challenges and problems they encountered, and what features they saw as valuable for BERT resources and educational materials.

**Development Stage 1 — January–March 2021**

Based on our opening brainstorming session, we began to develop tutorial materials that introduced humanities scholars to BERT conceptually as well as technically. We created a detailed slide presentation and two cloud-based code notebooks using Google Colab that walked participants through how to install, load, and apply BERT to collections of poetry and book reviews as well as how to interpret the results.

**Workshop 1 (By Invitation Only) — March 25, 2022**

We invited our DH-NLP Advisory Team and about 20 other relevant DH researchers to attend a 2-hour virtual BERT workshop and to serve as an initial focus group for our public-facing workshop later in June. Afterward, we solicited participants’ feedback and suggestions about the workshop via Zoom and over email.

**Development Stage 2 — March–May 2021**

Based on this workshop and participants’ feedback, we continued developing and refining our tutorial materials. We added a new code-based tutorial for working with Spanish-language sonnets, and we redesigned our slide presentation to foreground humanities use cases and to further emphasize intuitions alongside
technical explanations. We built a website for our project, and we also advertised our upcoming June workshop on the website, on social media, and over email.

**Workshop 2 (Free and Open to the Public) — June 4, 2021**

On June 4, 2021, we hosted a virtual BERT workshop that was free and open to the public. More than 90 participants attended the workshop.

**Public Release — June 2021**

We released our code-based tutorials and materials on our website: [http://www.bertforhumanists.org/](http://www.bertforhumanists.org/)

**Project Outcomes**

The initial phases of the project identified a number of issues faced by early adopters. We selected a few key themes and developed materials and explanations around these issues. Our final presentation consists of about two hours of introductory slides-based presentation and code examples.

**Infrastructure**

Using large language models often requires special hardware and computational infrastructure, which is a big obstacle for humanities scholars. LLMs require large numbers of matrix and tensor operations that are amenable to powerful dedicated parallel processing systems, such as graphics processing units (GPUs). But GPUs — specialized electronic circuits that are sometimes used by gamers and video editors — are not often found in a typical laptop or computer used by an academic. Additionally, the graphics hardware in the most common Intel-based Apple Macbooks are not compatible with the most common machine learning libraries.

Another serious and recurring problem for humanities researchers who want to use LLMs is known as “package management.” Python has emerged as the dominant programming language for general purpose machine learning, and Python “packages,” or software libraries, are vitally important for complicated machine learning routines. But Python is also notorious for the difficulty that comes with
managing third-party packages. Each Python machine learning package depends on specific versions of other packages, which in turn depend on other packages, all of which are under active development leading to mutual incompatibilities. There are multiple systems that exist for managing package dependencies (e.g., conda or pip) as well as for managing “virtual environments” that allow one program to use one version of a package and another program to use a different version. But using these systems and solving these package management problems are irritations even for experienced users. For new users, they’re far worse. The result is that excited and curious researchers are often confronted with an experience that can only be described as Kafkaesque, where even asking for help results in a barrage of unhelpfulness.

Our favored solution to all of these infrastructural problems is cloud-based notebooks, specifically the Google Colab system. Colab notebooks start with most of the packages needed for BERT pre-loaded, and adding missing packages is relatively quick and easy. Colab also offers hardware accelerators such as GPUs for free, and offers paid options for even higher performance. While the decision to use Colab means that we are dependent on a commercial service that could go away or increase its fees at any time, we have found that this system best meet the needs of our users.

Finally, another important infrastructure choice for our project is the use of the Huggingface “transformers” Python library as the primary library for our examples. Huggingface has an excellent Application Programming Interface (API), which, while still offering many challenges to novice users, is our preferred environment for working with LLMs. Huggingface not only offers a programming environment but also a hosting service for BERT models and their variants.

Machine learning concepts

During our initial brainstorming sessions, it became clear that successfully applying LLMs requires considerable knowledge about basic machine learning procedures and terminology and that these may not be familiar to most humanities researchers. While we ultimately determined that many of these concepts were

---

1 https://Colab.research.google.com/ Full disclosure: Mimno served as a Visiting Researcher at Google during this project, but we had previously used the system prior to this appointment.
beyond the scope of the tutorial, this decision limited our audience to those who had some experience with basic machine learning workflows. Properly introducing machine learning concepts from a humanities perspective would significantly increase the potential audience for LLM-based work.

We identified a small number of concepts that are vital to applying BERT to humanities material, which are related to workflows, training models, and evaluating results. We believe that we can develop tutorials that make these concepts accessible to broad audiences, but we chose not to do so for this project in the interest of time. These include data collection and curation, splitting data into training and held-out testing sets, making predictions from inputs to outputs, and evaluating predictions using metrics such as precision, accuracy, and F1 measures.

BERT-specific concepts
In addition to general machine learning concepts, the advisory panel assisted us in identifying concepts that are new to BERT and other LLMs, and that would not be familiar to researchers even with some exposure to machine learning. We focused on explaining and demystifying these concepts both in our "live" materials and through glossaries available on our project website: https://www.bertforhumanists.org/. It is important to note that there were a number of BERT-specific concepts that we did not choose to cover, as we felt they were sufficiently internal to the workings of the model, and that the cognitive load needed to explain them would not translate into greater usefulness for researchers. These concepts, such as multi-head attention and batch normalization, are also well-described by existing tutorial materials.

We identified the following topics as our most important priorities:

- Sub-word tokenization. NLP systems have traditionally operated by mapping words to integers in a vocabulary. In order to reduce vocabulary sizes, BERT and other LLMs have a numeric representation for more frequent words, but they break less frequent words into multiple sub-word tokens. For example, the word "pooling" might be represented as two sub-tokens "pool" and "-ing". This representation can sometimes result in surprising differences between how we perceive a word and how BERT handles it.
• Sequence-length limits. BERT is powerful because it represents the meaning of words in the context in which they appear in a sentence or passage, but this power comes at a cost: BERT cannot represent sequences longer than 512 sub-tokens. Participants need to be aware of this limitation and figure out how to map their research questions into spans of this length.

• Contextualized embeddings. Many DH researchers have become familiar over the past ten years with numeric vector representations of word types (that is, words as abstract dictionary entries), but BERT operates at the level of word tokens (individual instances of a string in a specific location in a sentence). Participants need to become comfortable with the idea that two examples of the same word may have different vector representations because they occur in a different context. One of the most consistently confusing and error-prone factors of contemporary machine learning is keeping track of the dimensions of data structures. In this case, we need to know that we are mapping a string into a sequence of vectors, one for each sub-word token.

• Pre-training and fine-tuning. Traditional machine learning with gradient-based optimization has one training phase. BERT and other "foundation" models (Bommasani et al. 2021) have two types of training: general-purpose pre-training and task-specific fine-tuning. Participants need to recognize the role of these steps and the situations in which the pre-trained model is sufficient and when additional fine-tuning is necessary.

• Multi-layer architectures. While we chose not to go into some of the details of how BERT works, such as attention, we determined that in order to use contextualized embeddings users needed to understand the way BERT operates iteratively through a series of layers. This representation of a series of layers that each contain a vector representation of each sub-word token helps explain what we are doing when we extract contextualized token embeddings.

Use Cases
In our presentation, we alternate between describing concepts using diagrams and illustrative analogies and demonstrating those concepts in worked Colab notebook examples. We selected two primary examples of applications that we thought were a good balance between being immediately compelling as tools for humanistic
inquiry and being sufficiently simple that we could explain them within fifteen to twenty minutes — word similarity with a collection of poetry and genre classification with a collection of online book reviews.

For our initial example, we created an interactive data visualization of a collection of poetry, in which selected words are mapped in space based on their similarity to one another (Figure 1). This is an unsupervised exploratory approach for studying word senses that conveniently requires no labeled data and works immediately from pre-trained models. Because BERT creates geometric representations of the contextual meaning of words, we can visualize those relationships by gathering contextual vectors for a large number of words, selecting a specific subset, and projecting their high-dimensional vectors to a 2D presentation. Participants found this view of word sense as position in a continuous space rather...
than as discrete dictionary senses both readily understandable and fascinating. We show that we can easily distinguish etymologically unrelated meanings of the same word by looking for well-separated clusters: in our collection of poetry, the word art as a noun is clearly distinct from its uses as a second-person verb (Figure 2). More subtle distinctions in the usage of a word may appear as smoother transitions or overlapping regions of density, with no clear hard boundary between senses. In one memorable example, we were able to extend the well-known word-sense distinction between a "money bank" and a "river bank" when the model led us to a third cluster associated with "fog bank".

In the second half of our tutorial, we presented a text classification example that requires fine-tuning. We specifically fine-tuned a BERT model on a dataset of Goodreads reviews, and we then used this fine-tuned model to predict the genre of books based on their Goodreads reviews. Before showing participants how to classify book genres with BERT, we showed them how to do so using the more basic method of logistic regression. We wanted to communicate to our participants that there are some research questions that can be answered with simpler methods and that it is always best practice to try such methods first. We also wanted to communicate the significance of establishing a baseline so that a researcher can assess whether BERT makes a meaningful contribution to their research. For our Goodreads task, we were able to show that BERT did make a meaningful improvement in the accuracy of classifying Goodreads reviews, and we were able to explore correct classifications as well as misclassifications by examining specific reviews and visualizing the data with heatmaps (Figure 3).
Multilingual modeling

While we include examples in English because it is the medium of the tutorial and of the original BERT model, digital humanities research is of course not limited to the English language. We therefore added an example using a Spanish-language BERT-style model available through the Huggingface library, and we replicated our word-sense disambiguation example using Spanish Golden Age sonnets.

Project Evaluation and Impact

The free virtual workshop that we hosted in June 2021 was advertised to and successfully solicited a broad audience of participants from a range of humanities disciplines, technical backgrounds, and institutions. For example, our social media advertisement of the workshop received more than 100 retweets, 200 likes, and 46,345 impressions on Twitter. In the end, more than 200 researchers from institutions around the world registered for the workshop, and more than 90 actually attended the event in June.

Beyond this workshop, we are pleased that the resources created by our project have also been shared, discussed, and taught widely. In November 2021, our project materials were highlighted in a blog post on DARIAH’s OpenMethods platform, where their editorial team selects leading digital tools proposed by “Community Volunteers” or chosen by the editors themselves. In May 2022, the BERT for Humanists projects was also spotlighted as a key machine learning resource in a blog post from Princeton’s Center for Digital Humanities. Testifying to its success as a teaching tool within the scholarly community, the BERT for Humanists Project was nominated for the “Best Digital Humanities Training Materials” award as part of the 2021 DH Awards.

Our BERT for Humanists resources have also proved useful for undergraduate and graduate students. They have been adopted and taught by instructors at universities across the country, such as in Ted Underwood’s 2021 “Data Science in the Humanities” course, taught in the Information School at the University of Illinois Urbana-Champaign. On social media, Underwood reported that this was a “good ex[ample] of @NEH_ODH impact” since “students are likely to use this outside the univ.” The tutorials we designed were also used in Lauren’s Klein’s 2021 “Practical Approaches to Data Science with Text” course at Emory University and in
Warut Khern-am-nuai’s “Data Mining for Business Analytics” course at McGill University.

Lastly, we presented many of the lessons that we learned from designing these tutorial materials at the 2021 Association for Computational Humanities (ACH) in a talk titled “BERT for Humanists: Uses, Challenges, Translations.”

**Project Continuation and Long-Term Impact**

After running our initial tutorials, we have received several invitations to run sequels and variations of the original tutorials. For example, on June 6, 2022, we ran a tutorial at the International Conferences for Web and Social Media (ICWSM) titled “BERT for Social Scientists and Humanists,” in which we updated our poetry-focused tutorial to include social media data, specifically Sephora makeup reviews. In a similar vein, we also presented an updated version of the tutorial for computational social scientists in December 2021 as part of the online school NLP+CSS. This tutorial was shared as a recorded YouTube video, and it has received more than 600 views.

Because of this demand and the many fascinating questions raised during our grant period, we are planning to expand our work with the BERT for Humanists project in the years to come, and we are currently developing an application for a Level III Digital Humanities Advancement Grant to support this work. Some of the questions that our workshop participants raised that we would like to continue addressing in the future are related to models and methods for different languages, especially low-resources languages, and the kinds of preprocessing that are required by BERT and similar LLMs, as well as whether and when this preprocessing makes sense for specific humanities use cases. Additionally, we want to consider newer text generation and text-to-text models such as GPT-3 and T5, which are paradoxically both much more complicated than BERT and much more accessible. Rather than using task-specific modules, these methods increasingly rely solely on text inputs and outputs, allowing users to essentially “talk” to models. Lastly, we would like to further consider the ethical considerations and possible harms of using LLMs, which is an important and thriving conversation in the machine learning
community, and we would like to enable humanities scholars to more fully be a part of it.

Works Cited


Appendix
Measuring Word Similarity with BERT (English Language Public Domain Poems)

By The BERT for Humanists Team

How can we measure the similarity of words in a collection of texts? For example, how similar are the words "nature" and "science" in a collection of 16th-20th century English language poems? Do 20th-century poets use the word "science" differently than 16th-century poets? Can we map all the different uses and meanings of the word "nature"?

The short answer is: yes! We can explore all of these questions with BERT, a natural language processing model that has revolutionized the field.

BERT turns words or tokens into vectors — essentially, a list of numbers in a coordinate system (x, y). We can then use the geometric similarity between these resulting vectors as a way to represent varying types of similarity between words.

In This Notebook

In this Colab notebook, we will specifically analyze a collection of poems scraped from Public-Domain-Poetry.com with the DistilBert model and the HuggingFace Python library. DistilBert is a smaller — yet still powerful! — version of BERT. By using the rich representations of words that BERT produces, we will then explore the multivalent meanings of particular words in context and over time.

We hope this notebook will help illustrate how BERT works, how well it works, and how you might use BERT to explore the similarity of words in a collection of texts. It is surprising, for example, that BERT works as well as it does, without any fine-tuning, on poems that were published hundreds of years before the text data it was trained on (Wikipedia pages and self-published novels).

But we also hope that these results will expose some of the limitations and challenges of BERT. We have to disregard poetic line breaks, for example, and we see that BERT has trouble with antiquated words like "thine," which don't show up in its contemporary vocabulary.

BERT Word Vectors: A Preview

Show code
The plot above displays a preview of our later results. This is what we’re working toward!

You can hover over each point to see the instance of each word in context. If you press shift and click on a point, you will be taken to the original poem on Public-Domain-Poetry.com. Try it out!

### Import necessary Python libraries and modules

Ok enough introduction! Let’s get started.

To use the HuggingFace transformers **Python library**, we first need to install it with pip.

```bash
!pip install transformers
```

```
Collecting transformers
  Downloading https://files.pythonhosted.org/packages/d5/43/cfe4ee779b6a678ac62ff5156ce2b91f
Collecting tokenizers<0.11,>=0.10.1
  Downloading https://files.pythonhosted.org/packages/d4/e2/df3543e8ffdad68f5acc6d64b823e175f5107952f51e3d22bb555e04
```
Then we will import the DistilBertModel and DistilBertTokenizerFast from the Hugging Face transformers library. We will also import a handful of other Python libraries and modules.

```python
# For BERT
from transformers import DistilBertTokenizerFast, DistilBertModel

# For data manipulation and analysis
import pandas as pd
pd.options.display.max_colwidth = 200
import numpy as np
from sklearn.decomposition import PCA

# For interactive data visualization
import altair as alt
```

Load text dataset

Our dataset contains around ~30 thousand poems scraped from [http://public-domain-poetry.com/](http://public-domain-poetry.com/). This website hosts a curated collection of poems that have fallen out of copyright, which makes them easier for us to share on the web. You can find the data in our [GitHub repository](https://github.com/).

We don't have granular date information about when each poem was published, but we do know the birth dates of most of our authors, which we've used to loosely categorize the poems by time.
period. The poems in our data range from the Middle Ages to the 20th Century, but most come from the 19th Century. The data features both well-known authors — William Wordsworth, Emily Elizabeth Dickinson, Paul Laurence Dunbar, Walt Whitman, Shakespeare — as well as less well-known authors.

Below we will use the Python library `pandas` to read in our CSV file of poems. It is convenient (especially for Colab notebooks) that `pandas` allows you to read in files directly from the web.

To be clear, however, knowledge of `pandas` is not necessary to use BERT. This is simply how we've chosen to load our data. All you really need is a list of texts (poems, passages, etc.). You can create this list however you are most comfortable.

```python
url = "https://raw.githubusercontent.com/melaniewalsh/BERT-4-Humanists/main/data/publi

poetry_df = pd.read_csv(url, encoding='utf-8')
# Show 5 random rows
poetry_df.sample(5)

<table>
<thead>
<tr>
<th>author</th>
<th>title</th>
<th>text</th>
</tr>
</thead>
</table>
| Fannie Isabelle Sherrick   | Two Pictures              | A beautiful form and a beautiful face,
A winsome bride and a woman's grace,
So fair and sweet it were heaven indeed
For man to follow where she would lead.
A web of lace and a jewel... |
| Arthur Hugh Clough         | Mari Magno or Tales on Board | A youth was I. An elder friend with me,
'Twas in September o'er the autumnal sea
We went; the wide Atlantic ocean o'er Two amongst many the strong steamer bore.
Delight it was to feel ... |
| Robert Lee Frost           | The Flood                 | Blood has been harder to dam back than water.
Just when we think we have it impounded safe
Behind new barrier walls (and let it chafe!),
It breaks away in some new kind of slaughter... |
```
Let's check to see how many poems are in this dataset:

```python
len(poetry_df)
```

```
31080
```

Let's check to see which authors show up the most in this dataset to get a sense of its contours:

```python
poetry_df['author'].value_counts()[:20]
```

```
Robert Herrick        1464
Madison Julius Cawein 1345
William Wordsworth   963
Thomas Moore          853
Thomas Hardy          655
Rudyard Kipling       638
Robert Burns          499
John Greenleaf Whittier 481
Algernon Charles Swinburne 461
Emily Elizabeth Dickinson 447
Paul Laurence Dunbar  417
John Clare            382
William Butler Yeats  378
Francesco Petrarca (Petrarch) 375
Paul Cameron Brown   341
Walt Whitman          338
Edgar Lee Masters     331
Percy Bysshe Shelley  330
Walter De La Mare     329
Oliver Wendell Holmes 329
Name: author, dtype: int64
```

Let's check to see what time periods show up the most in this dataset to get a sense of its contours:

```python
# Sort values, then create a histogram, and define the size of the figure
poetry_df['period'].sort_values().hist(figsize=(15, 5))
```
Sample text dataset

Though we wish we could analyze all the poems in this data, Colab tends to crash if we try to use more than 4-5,000 poems — even with DistilBert, the smaller version of BERT. This is an important limitation to keep in mind. If you'd like to use more text data, you might consider upgrading to a paid version of Colab (with more memory or GPUs) or using a compute cluster.

To reduce the number of poems, we will take a random sample of 1,000 poems from four different time periods: the 20th Century, 19th Century, 18th Century, and the Early Modern period.

# Filter the DataFrame for only a given time period, then randomly sample 1000 rows
nineteenth_sample = poetry_df[poetry_df['period'] == '19th Century'].sample(1000)
twentieth_sample = poetry_df[poetry_df['period'] == '20th Century'].sample(1000)
eighteenth_sample = poetry_df[poetry_df['period'] == '18th Century'].sample(1000)
sixteenth_sample = poetry_df[poetry_df['period'] == '16th-17th Centuries (Early Modern)'].sample(1000)

# Merge these random samples into a new DataFrame
poetry_df = pd.concat([sixteenth_sample, eighteenth_sample, twentieth_sample, nineteenth_sample])

poetry_df['period'].value_counts()

<table>
<thead>
<tr>
<th>Period</th>
<th>Count</th>
</tr>
</thead>
<tbody>
<tr>
<td>16th-17th Centuries (Early Modern)</td>
<td>1000</td>
</tr>
<tr>
<td>20th Century</td>
<td>1000</td>
</tr>
<tr>
<td>19th Century</td>
<td>1000</td>
</tr>
<tr>
<td>18th Century</td>
<td>1000</td>
</tr>
</tbody>
</table>

Finally, let's make a list of poems from our Pandas DataFrame.

poetry_texts = poetry_df['text'].tolist()
Let's examine a poem in our dataset:

```python
len(poetry_texts)
4000

print(poetry_texts[0])
When to thy porch I come and ravish'd see
The state of poets there attending thee,
Those bards and I, all in a chorus sing:
We are thy prophets, Porter, thou our king.
```

**Encode/tokenize text data for BERT**

Next we need to transform our poems into a format that BERT (via Huggingface) will understand. This is called *encoding or tokenizing* the data.

We will tokenize the poems with the `tokenizer()` from HuggingFace's DistilBertTokenizerFast. Here's what the `tokenizer()` will do:

1. Truncate the texts if they're more than 512 tokens or pad them if they're fewer than 512 tokens. If a word is not in BERT's vocabulary, it will be broken up into smaller "word pieces," demarcated by a `##`.

2. Add in special tokens to help BERT:
   - `[CLS]` — Start token of every document
   - `[SEP]` — Separator between each sentence
   - `[PAD]` — Padding at the end of the document as many times as necessary, up to 512 tokens
   - `##` — Start of a "word piece"

Here we will load DistilBertTokenizerFast from HuggingFace library, which will help us transform and encode the texts so they can be used with BERT.

```python
from transformers import DistilBertTokenizerFast

tokenizer = DistilBertTokenizerFast.from_pretrained('distilbert-base-uncased')
```
The `tokenizer()` will break word tokens into word pieces, truncate to 512 tokens, and add padding and special BERT tokens.

```python
from transformers import DistilBertModel

model = DistilBertModel.from_pretrained('distilbert-base-uncased').to('cuda')
```

Load pre-trained BERT model

Here we will load a pre-trained BERT model. To speed things up we will use a GPU, but using GPU involves a few extra steps. The command `to("cuda")` moves data from regular memory to the GPU's memory.
Get BERT word embeddings for each document in a collection

- This IS NOT expected if you are initializing DistilBertModel from the checkpoint.

To get word embeddings for all the words in our collection, we will use a `for` loop.

For each poem in our list `poetry_texts`, we will tokenize the poem, and we will extract the vocabulary word ID for each word/token in the poem (to use for later reference). Then we will run the tokenized poem through the BERT model and extract the vectors for each word/token in the poem.

We thus create two big lists for all the poems in our collection — `doc_word_ids` and `doc_word_vectors`.

```python
# List of vocabulary word IDs for all the words in each document (aka each poem)
doc_word_ids = []

# List of word vectors for all the words in each document (aka each poem)
doc_word_vectors = []

# Below we will slice our poem to ignore the first (0th) and last (-1) special BERT tokens
start_of_words = 1
end_of_words = -1

# Below we will index the 0th or first document, which will be the only document, since first_document = 0

for i, poem in enumerate(poetry_texts):
    # Here we tokenize each poem with the DistilBERT Tokenizer
    inputs = tokenizer(poem, return_tensors="pt", truncation=True, padding=True)

    # Here we extract the vocabulary word ids for all the words in the poem (the first
    # We ignore the first and last special BERT tokens
    # We also convert from a Pytorch tensor to a numpy array
    doc_word_ids.append(inputs.input_ids[first_document].numpy()[start_of_words:end_of_words])

    # Here we send the tokenized poems to the GPU
    # The model is already on the GPU, but this poem isn't, so we send it to the GPU
    inputs.to("cuda")

    # Here we run the tokenized poem through the DistilBERT model
    outputs = model(**inputs)

    # We take every element from the first or 0th document, from the 2nd to the 2nd to
    # Grabbing the last layer is one way of getting token vectors. There are different
Confirm that we have the same number of documents for both the tokens and the vectors:

```
len(doc_word_ids), len(doc_word_vectors)
```

```python
(4000, 4000)
```

```
doc_word_ids[0], doc_word_vectors[0]
```

```
(array([[ 2043,  2000, 15177,  7424,  1045,  2272,  1998, 16806,  4095,
            1005,  1040,  2156,  1996,  2110,  1997,  9736,  2045,  7052,
            14992,  1010,  2216,  22759,  2015,  1998, 1045, 1010,  2035,
            1999,  1037,  7165,  6170, 1024,  2057,  2024, 15177, 23172,
            1010,  8716,  1010, 15223,  2256,  2332,  1012]),
   array([[-0.4132233 ,  0.23495656,  0.352908  , ...,  0.4033177 ,
          0.24176075,  0.11400958,  0.4033177 , ...,  0.4033177 ,
          0.16030155,  0.658571  ,  0.46676674, ...,  0.658571  ,
          0.5922374 ,  0.14052588],
          [ 0.3989723 ,  0.69029903,  0.34327012, ...,  0.69029903,
          0.24893996, -0.24432279],
          [ 0.05397775,  0.2744551 ,  0.15093248, ..., -0.0344663 ,
          0.01873861, -0.4427815 ],
          [ 0.6110248 ,  0.45612204, -0.432807 , ...,  0.1745699 ,
          -0.42640686, -0.60329074]], dtype=float32))
```
# Dividing every vector by its length
all_word_vectors /= row_norms[:,np.newaxis]

- **Find all word positions in a collection**

We can use the array `all_word_ids` to find all the places, or *positions*, in the collection where a word appears.

We can find a word's vocab ID in BERT with `tokenizer.vocab` and then check to see where/how many times this ID occurs in `all_word_ids`.

```python
def get_word_positions(words):
    # This function accepts a list of words, rather than a single word

    # Get word/vocabulary ID from BERT for each word
    word_ids = [tokenizer.vocab[word] for word in words]

    # Find all the positions where the words occur in the collection
    word_positions = np.where(np.isin(all_word_ids, word_ids))[0]

    return word_positions
```

Here we'll check to see all the places where the word "bank" appears in the collection.

```python
generate_word_positions(["bank"])
```

- **Find word from word position**

Nice! Now we know all the positions where the word "bank" appears in the collection. But it would be more helpful to know the actual words that appear in context around it. To find these context words, we have to convert position IDs back into words.
Here we create an array so that we can go backwards from numeric token IDs to words:

```python
word_lookup = np.empty(tokenizer.vocab_size, dtype="O")

for word, index in tokenizer.vocab.items():
    word_lookup[index] = word
```

Now we can use `word_lookup` to find a word based on its position in the collection:

```python
word_positions = get_word_positions(["bank"])

for word_position in word_positions:
    print(word_position, word_lookup[all_word_ids[word_position]])
```

```
44625  bank
72923  bank
112668 bank
136834 bank
146050 bank
161849 bank
174665 bank
175830 bank
210853 bank
236842 bank
263222 bank
278202 bank
316528 bank
324283 bank
329082 bank
349864 bank
390063 bank
392736 bank
398382 bank
398391 bank
400587 bank
404543 bank
409092 bank
413181 bank
418273 bank
440440 bank
446953 bank
508341 bank
527366 bank
537401 bank
551233 bank
570985 bank
588659 bank
635716 bank
636715 bank
688100 bank
731126 bank
763534 bank
782510 bank
```
We can also look for the 3 words that come before "bank" and the 3 words that come after it.

```python
word_positions = get_word_positions(['bank'])

for word_position in word_positions:
    # Slice 3 words before "bank"
    start_pos = word_position - 3
    # Slice 3 words after "bank"
    end_pos = word_position + 4

    context_words = word_lookup[all_word_ids[start_pos:end_pos]]
    # Join the words together
    context_words = ' '.join(context_words)
    print(word_position, context_words)
```

```
44625 here by this bank of lil ##ies
72923 a sunni ##e bank ##e outstretched lay
112668 . on a bank , beside a
136834 by , this bank ##e with roses
146050 upon a green bank yielding room for
161849 , the blushing bank is all my
174665 ##no ' s bank , and on
175830 to the warm bank below , yellow
210853 ##s fra ##e bank to bra ##e
236842 ! from what bank came those live
263222 i . now bank an ' bra
278202 cows ##lip - bank and shady willow
316528 " see this bank - note -
324283 and the level bank of the swift
329082 river ' s bank , in fable
349864 , and the bank where they grew
390063 down upon a bank , where love
392736 sunshine on the bank : no tear
398382 " on a bank of flowers .
398391 . on a bank of flowers ,
400587 thames ' s bank , a young
404543 bee , from bank to bow ##er
409092 from the snowy bank those foot ##marks
413181 a thy ##my bank , and viewed
```
Let's make some functions that will help us get the context words around a certain word position for whatever size window (certain number of words before and after) that we want.

The first function `get_context()` will simply return the tokens without cleaning them, and the second function `get_context_clean()` will return the tokens in a more readable fashion.

```python
def get_context(word_id, window_size=10):
    """Simply get the tokens that occur before and after word position"""

    start_pos = max(0, word_id - window_size) # The token where we will start the context
    end_pos = min(word_id + window_size + 1, len(all_word_ids)) # The token where we will stop

    # Make a list called tokens and use word_lookup to get the words for given token IDs
    tokens = [word_lookup[word] for word in all_word_ids[start_pos:end_pos] ]

    context_words = " ".join(tokens)

    return context_words

import re

def get_context_clean(word_id, window_size=10):
```

"""Get the tokens that occur before and after word position AND make them more readable.

```
keyword = word_lookup[all_word_ids[word_id]]
start_pos = max(0, word_id - window_size) # The token where we will start the context
end_pos = min(word_id + window_size + 1, len(all_word_ids)) # The token where we will end the context

# Make a list called tokens and use word_lookup to get the words for given token IDs
tokens = [word_lookup[word] for word in all_word_ids[start_pos:end_pos]]

# Make wordpieces slightly more readable
# This is probably not the most efficient way to clean and correct for weird spacing
context_words = " ".join(tokens)
context_words = re.sub(r'\s+([##])', r'\1', context_words)
context_words = re.sub(r'\s+\s+', ' ', context_words)
context_words = re.sub(r'\s+s', 's', context_words)
context_words = re.sub(r'\s+s+', 's+', context_words)
context_words = re.sub(r'\s+\d', '\d', context_words)
context_words = re.sub(r'\s+\d+', '\d+', context_words)
context_words = re.sub(r'\s+’s', '’s', context_words)
context_words = re.sub(r'\s+’d', '’d', context_words)
context_words = re.sub(r'\s+’er', '’er', context_words)
context_words = re.sub(r'\s+\ ’', r'\ ’', context_words)
context_words = re.sub(r'\s+’s', '’s', context_words)
context_words = re.sub(r'\s+’d', '’d', context_words)

# Bold the keyword by putting asterisks around it
if keyword in context_words:
    context_words = re.sub(r'\b{keyword}\b', r'**{keyword}**', context_words)
    context_words = re.sub(r'\b{keyword} [esdtrlying]+\b', fr'**{keyword}**', context_words)

return context_words
```

To visualize the search keyword even more easily, we're going to import a couple of Python modules that will allow us to output text with bolded words and other styling. Here we will make a function `print_md()` that will allow us to print with Markdown styling.

```
from IPython.display import Markdown, display

def print_md(string):
    display(Markdown(string))

word_positions = get_word_positions(['bank'])

for word_position in word_positions:
    print_md(f"<br> {word_position}: {get_context_clean(word_position)} <br>")
```
44625: illis, rest but a while here by this bank of lilies; and lend a gentle ear to

72923: shore of muddie nile, upon a sunnie banke outstretched lay, in monstrous length, a might

112668: old theatres, and build up new. on a bank, beside a willow, heaven her covering, earth

136834: , all our flocks are feeding by, this banke with roses spred, oh it is a

146050: now as an angler melancholy standing upon a green bank yielding room for landing, a wriggling yelk

161849: at th'shepherd's nose, the blushing bank is all my care, with hearth so red,

174665: 's golden store, on arno's bank, and on that bloomy shore, warbling

175830: ample numbers stray. ii. then to the warm bank below, yellow with the morning-ray, and

210853: the burn comes down, and roars frae bank to brae; and bird and beast in covert

236842: proof, though christian rites be wanting! from what bank came those live herbs? by what hand were

263222: ye know that he is just. i. now bank an 'brae are claih'd in

278202: , in this sweet book, the cowslip-bank and shady willow-tree; and the fresh mead

316528: , a senator addressing, said: "see this bank-note-lo! a blessing-breathe on

329082: thus, on the river's bank, in fabled lore, the rustic stands;

349864: took a view of my favourite field, and the bank where they grew; and now in the grass behold

390063: some other ground. i laid me down upon a bank, where love lay sleeping; i heard among the

392736: the tree, the wood, the sunshine on the bank: no tear, no thought of time's

398382: -appear no more. tune-"on a bank of flowers. "i. on a bank of

398391: a bank of flowers. "i. on a bank of flowers, in a summer day, for summer

400587: friendly breast cecilia liv'd on thames's bank, a young and lovely married fair; to creatures

404543: and our own. thus as the bee, from bank to bower, assiduous sips at

Here we make a list of all the context views for our keyword.
Finally, we don't just want to read all the instances of "bank" in the collection, we want to measure the similarity of all the instances of "bank."

To measure similarity between all the instances of "bank," we will take the vectors for each instance and then use PCA to reduce each 768-dimensionsal vector to the 2 dimensions that capture the most variation.

Then, for convenience, we will put these PCA results into a Pandas DataFrame, which will use to generate an interactive plot.
illis, rest but a while here by this **bank** of lilies; and lend a gentle ear to
shore of muddie nile, upon a sunnie **banke** outstretched lay, in monstrous length, a might

### Match context with original text and metadata

- 6 cells hidden

### Plot word embeddings

Lastly, we will plot the words vectors from this DataFrame with the Python data viz library Altair.

```python
import altair as alt

alt.Chart(df,title="Word Similarity: Bank").mark_circle(size=200).encode(
    alt.X('x', scale=alt.Scale(zero=False)), y="y",
    # If you click a point, take you to the URL link
    href="link",
    # The categories that show up in the hover tooltip
    tooltip=['title', 'context', 'author', 'period']
).interactive().properties(width=500,
    height=500)
```
Plot word embeddings from keywords (all at once!)

We can put the code from the previous few sections into a single cell and plot the BERT word embeddings for any list of words. Let's look at the words "nature," "religion," "science," and "art."

```python
# List of keywords that you want to compare
keywords = ['nature', 'religion', 'science', 'art']

# How to color the points in the plot. The other option is "period" for time period
color_by = 'word'

# Get all word positions
word_positions = get_word_positions(keywords)

# Get all contexts around the words
keyword_contexts = []
keyword_contexts_tokens = []
words = []

for position in word_positions:
    words.append(word_lookup[all_word_ids[position]])
    keyword_contexts.append(get_context_clean(position))
    keyword_contexts_tokens.append(get_context(position))

# Reduce word vectors with PCA
pca = PCA(n_components=2)
pca.fit(all_word_vectors[word_positions,::].T)

# Make a DataFrame with PCA results
df = pd.DataFrame({"x": pca.components_[0,:], "y": pca.components_[1,:],
                   "context": keyword_contexts, "tokens": keyword_contexts_tokens, "w"})
```

# Match original text and metadata
df[['title', 'author', 'period', 'link']] = df.apply(find_original_poem, axis='columns')

# Rename columns so that the context shows up as the "title" in the tooltip (bigger ar
df = df.rename(columns={'title': 'poem_title', 'context': 'title'})

# Make the plot
alt.Chart(df, title=f"Word Similarity: {', '.join(keywords).title()}").mark_circle(siz
alt.X('x',
scale=alt.Scale(zero=False)
), y="y",
color= color_by,
href="link",
tooltip=['title', 'word', 'poem_title', 'author', 'period']
).interactive().properties(
width=500,
height=500
)
Let's examine the words "nature," "religion," "science," and "art" again but this time color the points by their time period.

# List of keywords that you want to compare
keywords = ['nature', 'religion', 'science', 'art']

# How to color the points in the plot
color_by = 'period'

# Get all word positions
word_positions = get_word_positions(keywords)

# Get all contexts around the words
keyword_contexts = []
keyword_contexts_tokens = []
words = []

for position in word_positions:
    words.append(word_lookup[all_word_ids[position]])
    keyword_contexts.append(get_context_clean(position))
    keyword_contexts_tokens.append(get_context(position))

# Reduce word vectors with PCA
pca = PCA(n_components=2)
pca.fit(all_word_vectors[word_positions, :].T)

# Make a DataFrame with PCA results
df = pd.DataFrame({'x': pca.components_[0, :], 'y': pca.components_[1, :],
                   'context': keyword_contexts, 'tokens': keyword_contexts_tokens, 'wo

# Match original text and metadata
df[['title', 'author', 'period', 'link']] = df.apply(find_original_poem, axis='columns:

# Rename columns so that the context shows up as the "title" in the tooltip (bigger ar
df = df.rename(columns={'title': 'poem_title', 'context': 'title'})

# Make the plot
alt.Chart(df, title=f"Word Similarity: {', '.join(keywords).title()}").mark_circle(si:
    alt.X('x',
        scale=alt.Scale(zero=False)
    ), y="y",
    color= 'period',
    href="link",
    tooltip=['title', 'word', 'poem_title', 'author', 'period'
    ]).interactive().properties(
        width=500,
        height=500
    )
Let's compare the words "mean," "thin," "average," and "cruel."

# List of keywords that you want to compare
keywords = ['mean', 'thin', 'average', 'cruel']

# How to color the points in the plot. The other option is "period" for time period
color_by = 'word'

# Get all word positions
word_positions = get_word_positions(keywords)

# Get all contexts around the words
keyword_contexts = []
keyword_contexts_tokens = []
words = []

for position in word_positions:
    words.append(word_lookup[all_word_ids[position]])
    keyword_contexts.append(get_context_clean(position))
    keyword_contexts_tokens.append(get_context(position))

# Reduce word vectors with PCA
pca = PCA(n_components=2)
pca.fit(all_word_vectors[word_positions, :].T)

# Make a DataFrame with PCA results
df = pd.DataFrame({'x': pca.components_[0, :], 'y': pca.components_[1, :],
                   'context': keyword_contexts, 'tokens': keyword_contexts_tokens, 'wo

# Match original text and metadata
df[['title', 'author', 'period', 'link']] = df.apply(find_original_poem, axis='columns

# Rename columns so that the context shows up as the "title" in the tooltip (bigger ar
df = df.rename(columns={'title': 'poem_title', 'context': 'title'})

# Make the plot
alt.Chart(df, title=f"Word Similarity: {', '.join(keywords).title()}").mark_circle(siz
  alt.X('x',
    scale=alt.Scale(zero=False)
  ), y="y",
  color= color_by,
  href="link",
  tooltip=['title', 'word', 'poem_title', 'author', 'period']
).interactive().properties(
  width=500,
  height=500
)
Let's compare the words 'head', 'heart', 'eye', 'arm', and 'leg.'

# List of keywords that you want to compare
keywords = ['head', 'heart', 'eye', 'arm', 'leg']

# How to color the points in the plot. The other option is "period" for time period
color_by = 'word'

# Get all word positions
word_positions = get_word_positions(keywords)

# Get all contexts around the words
keyword_contexts = []
keyword_contexts_tokens = []
words = []

for position in word_positions:
    words.append(word_lookup[all_word_ids[position]])
    keyword_contexts.append(get_context_clean(position))
    keyword_contexts_tokens.append(get_context(position))

# Reduce word vectors with PCA
pca = PCA(n_components=2)
pca.fit(all_word_vectors[word_positions, :].T)

# Make a DataFrame with PCA results
df = pd.DataFrame({
    "x": pca.components_[0,:], 
    "y": pca.components_[1,:], 
    "context": keyword_contexts, 
    "tokens": keyword_contexts_tokens, 
    "word": words, 
})

# Match original text and metadata
df[['title', 'author', 'period', 'link']] = df.apply(find_original_poem, axis='columns')

# Rename columns so that the context shows up as the "title" in the tooltip (bigger ar
da = df.rename(columns={'title': 'poem_title', 'context': 'title'})

# Make the plot
alt.Chart(df, title=f"Word Similarity: {', '.join(keywords).title()}"").mark_circle(size=alt.X('x', scale=alt.Scale(zero=False)), y="y", color=color_by, href="link", tooltip=['title', 'word', 'poem_title', 'author', 'period']).interactive().properties(width=500, height=500)
# List of keywords that you want to compare
keywords = ['ring']

# How to color the points in the plot. The other option is "period" for time period
color_by = 'word'

# Get all word positions
word_positions = get_word_positions(keywords)

# Get all contexts around the words
keyword_contexts = []
keyword_contexts_tokens = []
words = []

for position in word_positions:
    words.append(word_lookup[all_word_ids[position]])
    keyword_contexts.append(get_context_clean(position))
    keyword_contexts_tokens.append(get_context(position))
# Reduce word vectors with PCA
pca = PCA(n_components=2)
pca.fit(all_word_vectors[word_positions,:].T)

# Make a DataFrame with PCA results
df = pd.DataFrame({
    "x": pca.components_[0,:],
    "y": pca.components_[1,:],
    "context": keyword_contexts,
    "tokens": keyword_contexts_tokens,
})

# Match original text and metadata
df[,['title', 'author', 'period', 'link']] = df.apply(find_original_poem, axis='columns')

# Rename columns so that the context shows up as the "title" in the tooltip (bigger ar
df = df.rename(columns={'title': 'poem_title', 'context': 'title'})

# Make the plot
alt.Chart(df, title=f"Word Similarity: {', '.join(keywords).title()}").mark_circle(size=500,
scale=alt.Scale(zero=False),
   y="y",
   color= color_by,
   href="link",
   tooltip=["title", "word", "poem_title", "author", "period"]
).interactive().properties(
   width=500,
   height=500
)
Write to CSV

```python
df.to_csv('bert-word-ring.csv', index=False, encoding='utf-8')
```

### Find word similarity from a specific word position

We can also search all of the vectors for words similar to a query word.

```python
def get_nearest(query_vector, n=100):
    cosines = all_word_vectors.dot(query_vector)
    ordering = np.flip(np.argsort(cosines))
    return ordering[:n]
```

To do so, we need to find the specific word position of our desired search keyword.

```python
word_positions = get_word_positions(['bank'])

for word_position in word_positions:
    print_md(f"<br> {word_position}: {get_context_clean(word_position)} <br>")
```
illis, rest but a while here by this bank of lilies; and lend a gentle ear to

shore of muddie nile, upon a sunnie banke outstretched lay, in monstrous length, a might

old theatres, and build up new. on a bank, beside a willow, heaven her covering, earth

, all our flocks are feeding by, this banke with roses spred, oh it is a

now as an angler melancholy standing upon a green bank yielding room for landing, a wriggling yellow

at th'shepherd's nose, the blushing bank is all my care, with hearth so red,

's golden store, on arno's bank, and on that bloomy shore, warbling

ample numbers stray. ii. then to the warm bank below, yellow with the morning-ray, and

the burn comes down, and roars frae bank to brae; and bird and beast in covert

proof, though christian rites be wanting! from what bank came those live herbs? by what hand were they

ye know that he is just. i. now bank an 'brae are claith'd in

, in this sweet book, the cowslip-bank and shady willow-tree; and the fresh mead

a senator addressing, said: "see this bank-note-lo! a blessing-breathe on

ious greeks, swept the foundation, and the level bank of the swift-rolling hellespont restored

thus, on the river's bank, in fabled lore, the rustic stands;

the defendant discovered a widow with gold in the bank and the plaintiff was left in the cold. an

keyword_position = 897288

 contexts = [get_context_clean(token_id) for token_id in get_nearest(all_word_vectors{})

 for context in contexts:
   print_md(context)
, the defendant discovered a widow with gold in the bank and the plaintiff was left in the cold. an
in love with it; i will go to the bank by the wood, and become undisguised
, a senator addressing, said: "see this bank-note-lo! a blessing-breathe on
's golden store, on arno's bank, and on that bloomy shore, warbling
a crank that increases the balance at somebody's bank; and i feel satisfaction that mother is free from
the watchman climbs the stair... the bank defaulter leers at a chaos of figures,
lovely child grows white and white, as on the bank she lingers. "the law, my child
at th'shepherd's nose, the blushing bank is all my care, with hearth so red,
friendly breast cecilia liv'd on thames's bank, a young and lovely married fair; to creatures
and our own. thus as the bee, from bank to bower, assiduous sips at
s city, by a rotten tree, or woodland bank! in ignorance we muse: pausing, annoy
some other ground. i laid me down upon a bank, where love lay sleeping; i heard among the
illis, rest but a while here by this bank of lilies; and lend a gentle ear to
. how sweet, when weary, dropping on a bank, turning a look around on things that be!
the bridge they came. they followed from the snowy bank those footmarks, one by one, into the
the tree, the wood, the sunshine on the bank: no tear, no thought of time's
a goat need be-lay on a thymy bank, and viewed himself reflected in the flood. "
ed turned the crank, an 'there on the bank they squatted like bumps on a log. for
took a view of my favourite field, and the bank where they grew; and now in the grass behold
old theatres, and build up new. on a bank, beside a willow, heaven her covering, earth
that rank the slow brook's heron-haunted bank. the dragon-flies, brass-bright and
generals are putting on civvies and looking like bank clerks. public officials are getting friendly. the policeman
gossamers of silver lace, and the turf bank wears with glee black and silver filigree.
the burn comes down, and roars frae bank to brae; and bird and beast in covert
delions like to suns will bloom, aside some bank or hillock creeping low--;though each
eye know that he is just. i. now bank an 'brae are claith'd in
now as an angler melancholy standing upon a green bank yielding room for landing, a wriggling yellow
shore, where two contracted new come daily to the banks, that when they see return of love, more
wolf, and grapple with the bear. this bank, in which the dead were laid, was sacred
in the bank and the plaintiff was left in the cold. an hour smithers spoke, and he said
birth, couched like a king each on its bank of earth arbalist, manganel and cat
would simply not lose it. it would lie in banks and old stockings and kindred receptacles
i waited, and the sunshine flecked the bank happy with arbutus and violets where i
of a fir, the defendant discovered a widow with gold in the bank and the plaintiff was left in the
, in this sweet book, the cowslip-bank and shady willow-tree; and the fresh mead
ious greeks, swept the foundation, and the level bank of the swift-rolling hellespont restored
and the hollow tree for the buzzing bee and a bank for the wasp to hive in. vi. and
. there's barbara cowie, comely bank and may, christened, at home, in
? come, come away to the river's bank, come in the early morning; come when the
which our old traditions tell. for here the upland bank sends out a ridge toward the river-side;
mists that cloak hanger and hollied bank, the winter world awoke to hear the feeble
across a huge gulf. on the other bank crouches april with her hair as smooth and straight
across a huge gulf... on the other bank odalisque april with her hair as smooth and straight and rowland pays his balance, to catch the banker all have sought, but still the rogue yakur...
Training and Fine-Tuning BERT for Classification

Classifying Goodreads Reviews By Book Genre

By The BERT for Humanists Team

This notebook will demonstrate how users can train and fine-tune a BERT model for classification with the popular HuggingFace transformers Python library.

We will fine-tune a BERT model on Goodreads reviews from the UCSD Book Graph with the goal of predicting the genre of the book being reviewed. The genres include:

- poetry
- comics & graphic
- fantasy & paranormal
- history & biography
- mystery, thriller, & crime
- romance
- young adult

Basic steps involved in using BERT and HuggingFace:

1. Divide your data into training and test sets.
2. Encode your data into a format BERT will understand.
3. Combine your data and labels into dataset objects.
4. Load the pre-trained BERT model.
5. Fine-tune the model using your training data.
6. Predict new labels and evaluate performance on your test data.

Import necessary Python libraries and modules

First, we will import necessary Python libraries and modules. These include as gdown, for downloading large files from Google Drive (where we will get our UCSD Goodreads reviews), as well as scikit-learn (sklearn) and PyTorch (torch), for various machine learning tools.
To use the HuggingFace transformers Python library, we will install it with pip.

!pip3 install transformers

Collecting transformers
  Downloading https://files.pythonhosted.org/packages/d5/43/cfe4ee779bbd6a678ac6

Collecting huggingface-hub==0.0.8
  Downloading https://files.pythonhosted.org/packages/a1/88/7b1e45720ecf59c6c673
Collecting sacremoses
Collecting tokenizers<0.11,>=0.10.1

Once transformers is installed, we will import modules for DistilBert, a distilled or smaller version of a BERT model that runs more quickly and uses less computing power. This makes it ideal for those just getting started with BERT.

```python
from transformers import DistilBertTokenizerFast, DistilBertForSequenceClassification
```

**Set parameters and file paths**

```python
# This is the name of the BERT model that we want to use.
# We're using DistilBERT to save space (it's a distilled version of the full BERT model)
# and we're going to use the cased (vs uncased) version.
model_name = 'distilbert-base-cased'

# This is the name of the program management system for NVIDIA GPUs. We're going to use it.
device_name = 'cuda'

# This is the maximum number of tokens in any document sent to BERT.
max_length = 512

# This is the name of the directory where we'll save our model. You can name it whatever you like.
cached_model_directory_name = 'distilbert-reviews-genres'
```
Load and sample Goodreads data

In this cell, we create a Python dictionary with each genre and the link to the corresponding UCSD Goodreads review data for that genre on Google Drive.

*If you manually click on any of the URLs, you will be able to download the data for that genre. For example, here's the link for poetry: [https://drive.google.com/uc?id=1FVD3LxJXRc5GrKm97LehLgVGbRfF9TyO](https://drive.google.com/uc?id=1FVD3LxJXRc5GrKm97LehLgVGbRfF9TyO)*

```python
genre_url_dict = {
    'poetry': 'https://drive.google.com/uc?id=1FVD3LxJXRc5GrKm97LehLgVGbRfF9TyO',
    'children': 'https://drive.google.com/uc?id=1908GDMdrcTMz9Urnz7fXU17X2Zp3F0U',
    'comics_graphic': 'https://drive.google.com/uc?id=1V4MLeoEin7HcJGvIv3Ei0eTqczRqG9L7',
    'fantasy_paranormal': 'https://drive.google.com/uc?id=1THnnmE4X5zQzHsCjG7sJ9c7IiZDq2FJW',
    'history_biography': 'https://drive.google.com/uc?id=1lDkTzm6rEoIY8fXj5Wz8pWJQ9G0ZaF3',
    'mystery_thriller_crime': 'https://drive.google.com/uc?id=1ONpyuv0v5rWqY4GxZbIqD5rP25Y5ZgU',
    'romance': 'https://drive.google.com/uc?id=1NpFsDQKBH17X2Zp3F0U',
    'young_adult': 'https://drive.google.com/uc?id=1M5iqCZ8a2637FVWj5Wz8pWJQ9G0ZaF3'
}
```

Next we loop through this dictionary and use `gdown` to download the Goodreads review data for each genre from Google Drive.

```python
for _genre, _url in genre_url_dict.items():
    gdown.download(_url, _genre + '.json.gz', quiet=False)
```

```
Downloading...
From: https://drive.google.com/uc?id=1FVD3LxJXRc5GrKm97LehLgVGbRfF9TyO
To: /content/poetry.json.gz
49.3MB [00:01, 44.4MB/s]
```

```
Downloading...
From: https://drive.google.com/uc?id=1908GDMdrcTMz9Urnz7fXU17X2Zp3F0U
To: /content/children.json.gz
172MB [00:02, 78.2MB/s]
```

```
Downloading...
From: https://drive.google.com/uc?id=1V4MLeoEin7HcJGvIv3Ei0eTqczRqG9L7
To: /content/comics_graphic.json.gz
147MB [00:02, 64.4MB/s]
```

```
Downloading...
From: https://drive.google.com/uc?id=1THnnmE4X5zQzHsCjG7sJ9c7IiZDq2FJW
To: /content/fantasy_paranormal.json.gz
1.26GB [00:15, 80.2MB/s]
```

```
Downloading...
From: https://drive.google.com/uc?id=1lDkTzm6rEoIY8fXj5Wz8pWJQ9G0ZaF3
To: /content/history_biography.json.gz
0.72GB [00:21, 35.6MB/s]
```

```
Downloading...
From: https://drive.google.com/uc?id=1ONpyuv0v5rWqY4GxZbIqD5rP25Y5ZgU
To: /content/history_biography.json.gz
0.23GB [00:30, 28.0MB/s]
```

```
Downloading...
From: https://drive.google.com/uc?id=1NpFsDQKBH17X2Zp3F0U
To: /content/mystery_thriller_crime.json.gz
0.88GB [00:30, 30.1MB/s]
```

```
Downloading...
From: https://drive.google.com/uc?id=1M5iqCZ8a2637FVWj5Wz8pWJQ9G0ZaF3
To: /content/romance.json.gz
1.12GB [00:30, 32.3MB/s]
```

```
Downloading...
From: https://drive.google.com/uc?id=1lDkTzm6z8z9fGvIv3Ei0eTqczRqG9L7
To: /content/young_adult.json.gz
0.89GB [00:30, 31.6MB/s]
```
```
If you click the file browser icon in the left-hand side bar, you should see that these files have now been downloaded.

Next we create a function `load_reviews()`, which will use `gzip` to unzip the downloaded Goodreads review JSON files and `json` to load the JSON files once they're unzipped.

```python
def load_reviews(file_name, head=None):
    reviews = []
    count = 0

    with gzip.open(file_name) as file:
        for line in file:
            d = json.loads(line)
```
count += 1

_book_id = d['book_id']

reviews.append(d['review_text'])

# Break if we reach the Nth line
if (head is not None) and (count > head):
    break

return reviews

Now we apply the load_reviews() function. For each genre, we load and unzip the corresponding .json.gz file, e.g., poetry.json.gz, then we randomly sample 2000 Goodreads reviews and make a dictionary genre_reviews_dict of all these reviews.

genres = ['poetry', 'children', 'comics_graphic', 'fantasy_paranormal', 'history_biog
genre_reviews_dict = {}

for _genre in genres:
    print('Loading ' + _genre + '.json.gz')

    _reviews = load_reviews(_genre + '.json.gz')
    genre_reviews_dict[_genre] = random.sample(_reviews, 2000)

    Loading poetry.json.gz
    Loading children.json.gz
    Loading comics_graphic.json.gz
    Loading fantasy_paranormal.json.gz
    Loading history_biography.json.gz
    Loading mystery_thriller_crime.json.gz
    Loading romance.json.gz
    Loading young_adult.json.gz

Let's preview a couple of the key-value pairs in genre_reviews_dict

for _genre, _reviews in genre_reviews_dict.items():
    print(_genre)
    print(random.sample(_reviews, 1)[0])

    poetry
    Creme de la creme!
    For men may come and men may go,
    But I go on forever.
    A poem that is almost lyrical to the sound the flowing water makes as it winds
    The very purest form of poetry.
    children
    When I was in 7th grade, our English class was broken down into reading groups.
Then I never read it.
I always meant to, but there were other things like school, travel and college
This. Book. Is. Great! Starting out the writing style reminded me of the beginn
The main thing I loved about this book is the easy battle of light and dark. In s
More than just a comic book. The layers of the story, the emotion, and the actic
So all my friends were right. This book is amazing! This is a must read. Do it.
Another great set of Tragedies.
Philoctetes and the Bacchae were my favourites but all in all a great collectic
Incredible; fantastic - review coming soon!
Marrying Winterborne was going great until around half the book, where the hero
I know it was necessary for the conflict but I thought a writer of Ms. Kleypas'
Part of the problem was that a significant part of the setup occurred in Cold-H
The narration was good, except for Winteborne's voice, which was atrocious. I o
I don't want to say it, because I don't want it to be true, but I'm starting to
Boy meets girl. Girl meets boy. Oops, girl is a dragon who can shapeshift into a

Here we use pickle to save this Python dictionary to a .pickle file so we can easily load it later.

The pickle module allows you to save and load Python objects like lists and dictionaries.

```python
pickle.dump(genre_reviews_dict, open('genre_reviews_dict.pickle', 'wb'))
# genre_reviews_dict = pickle.load(open('genre_reviews_dict.pickle', 'rb'))
```

- **Split the data into training and test sets**

When training a machine learning model, it is necessary to split your training data into two parts: a "training" set and a "test" set.

We will train our BERT model on the "training" set of Goodreads reviews and then we will evaluate how well it is performing by running it on the "test" set of Goodreads reviews that the model has never seen before.

Normally, to tune the hyperparameters, you should also create a "validation" set for tuning, and only use the "test" set once, at the end of all tuning. For simplicity, in this tutorial, we will only using a training and test set.
train_texts = []
train_labels = []

test_texts = []
test_labels = []

for _genre, _reviews in genre_reviews_dict.items():
    _reviews = random.sample(_reviews, 1000)  # Use a very small set as an example.

    for _review in _reviews[:800]:
        train_texts.append(_review)
        train_labels.append(_genre)

    for _review in _reviews[800:]:
        test_texts.append(_review)
        test_labels.append(_genre)

Show how many Goodreads reviews and labels we have in each category: 6400 training reviews, 6400 training labels (genres), 1600 test reviews, 1600 test labels (genre)

len(train_texts), len(train_labels), len(test_texts), len(test_labels)

(6400, 6400, 1600, 1600)

Here's an example of a training label and review:

train_labels[0], train_texts[0]

('poetry',
 'I'll be honest--I've never read it straight from start to finish. But I've had

**Run a baseline model (logistic regression)**

Here we train and evaluate a simple TF-IDF baseline model using logistic regression.
We find better-than-random performance, even for a very small dataset. We'll see whether BERT can beat this good baseline!

vectorizer = TfidfVectorizer()
X_train = vectorizer.fit_transform(train_texts)
X_test = vectorizer.transform(test_texts)
We train a logistic regression model from scikit-learn on the Goodreads training data, and then we use the trained model to make predictions on our Goodreads review test set.

```python
model = LogisticRegression(max_iter=1000).fit(X_train, train_labels)
predictions = model.predict(X_test)
```

We can use scikit-learn's `classification_report` function to evaluate how well the logistic regression model's predictions match up with the true labels for the Goodreads reviews.

Importantly, we can see that our average scores are above random performance (we have 8 classes, so random performance would be ~0.2).

```bash
print(classification_report(test_labels, predictions))
```

<table>
<thead>
<tr>
<th></th>
<th>precision</th>
<th>recall</th>
<th>f1-score</th>
<th>support</th>
</tr>
</thead>
<tbody>
<tr>
<td>children</td>
<td>0.63</td>
<td>0.62</td>
<td>0.63</td>
<td>200</td>
</tr>
<tr>
<td>comics_graphic</td>
<td>0.60</td>
<td>0.62</td>
<td>0.61</td>
<td>200</td>
</tr>
<tr>
<td>fantasy_paranormal</td>
<td>0.34</td>
<td>0.26</td>
<td>0.29</td>
<td>200</td>
</tr>
<tr>
<td>history_biography</td>
<td>0.54</td>
<td>0.46</td>
<td>0.49</td>
<td>200</td>
</tr>
<tr>
<td>mystery_thriller_crime</td>
<td>0.50</td>
<td>0.48</td>
<td>0.49</td>
<td>200</td>
</tr>
<tr>
<td>poetry</td>
<td>0.60</td>
<td>0.79</td>
<td>0.68</td>
<td>200</td>
</tr>
<tr>
<td>romance</td>
<td>0.45</td>
<td>0.48</td>
<td>0.47</td>
<td>200</td>
</tr>
<tr>
<td>young_adult</td>
<td>0.38</td>
<td>0.38</td>
<td>0.38</td>
<td>200</td>
</tr>
</tbody>
</table>

<p>| | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>accuracy</td>
<td></td>
<td></td>
<td>0.51</td>
<td>1600</td>
</tr>
<tr>
<td>macro avg</td>
<td>0.50</td>
<td>0.51</td>
<td>0.50</td>
<td>1600</td>
</tr>
<tr>
<td>weighted avg</td>
<td>0.50</td>
<td>0.51</td>
<td>0.50</td>
<td>1600</td>
</tr>
</tbody>
</table>

**Encode data for BERT**

We're going to transform our texts and labels into a format that BERT (via Huggingface and PyTorch) will understand. This is called *encoding* the data.

Here are the steps we need to follow:

1. The labels—in this case, Goodreads genres—need to be turned into integers rather than strings.
2. The texts—in this case, Goodreads reviews—need to be truncated if they're more than 512 tokens or padded if they're fewer than 512 tokens. The tokens, or words in the texts, also need to be separated into "word pieces" and matched to their embedding vectors.
3. We need to add special tokens to help BERT:

<table>
<thead>
<tr>
<th>BERT special token</th>
<th>Explanation</th>
</tr>
</thead>
<tbody>
<tr>
<td>[CLS]</td>
<td>Start token of every document.</td>
</tr>
<tr>
<td>[SEP]</td>
<td>Separator between each sentence</td>
</tr>
<tr>
<td>[PAD]</td>
<td>Padding at the end of the document as many times as necessary, up to 512 tokens</td>
</tr>
<tr>
<td>##</td>
<td>Start of a &quot;word piece&quot;</td>
</tr>
</tbody>
</table>

Here we will load `DistilBertTokenizerFast` from the HuggingFace library, which will do all the work of encoding the texts for us. The `tokenizer()` will break word tokens into word pieces, truncate to 512 tokens, and add padding and special BERT tokens.

```python
tokenizer = DistilBertTokenizerFast.from_pretrained(model_name)  # The model_name need
```

Here we will create a map of our labels, or Goodreads genres, to integer keys. We take the unique labels, and then we make a dictionary that associates each label/tag with an integer.

**Note:** HuggingFace documentation sometimes refers to "labels" as "tags" but these are the same thing. We use "labels" throughout this notebook for clarity.

```python
unique_labels = set(label for label in train_labels)
label2id = {label: id for id, label in enumerate(unique_labels)}
id2label = {id: label for label, id in label2id.items()}
```

```python
label2id.keys()
```

```python
```

```python
id2label.keys()
```

```python
dict_keys([0, 1, 2, 3, 4, 5, 6, 7])
```

Now let's encode our texts and labels!
train_encodings = tokenizer(train_texts, truncation=True, padding=True, max_length=max)
test_encodings = tokenizer(test_texts, truncation=True, padding=True, max_length=max)

train_labels_encoded = [label2id[y] for y in train_labels]
test_labels_encoded = [label2id[y] for y in test_labels]

Examine a Goodreads review in the training set after encoding

'.join(train_encodings[0].tokens[0:100])

' [CLS] I ' ll be honest -- I ' ve never read it straight from start to finish.-us courses that I feel I ' ve got a decent handle of it. [SEP] [PAD] [PAD] [PAD] [PAD] [PAD] [PAD] [PAD] [PAD] [PAD] [PAD] [PAD] [PAD] [PAD] [PAD] [PAD]

Examine a Goodreads review in the test set after encoding

'.join(test_encodings[0].tokens[0:100])

' [CLS] l ' ` l ##m s ##h ` w ##ry t ##H ##dy ##dan m ##n l ##dy ##wn b ##sh ##k #h w ##lm ' s ##t ##T ` m b ` D ##h , w ##m z ##lt ' b ##H ##th ` n s ##h ` w ## ##S ##y ' G l t ##nt ##Z ##r s'

Examine the training labels after encoding

set(train_labels_encoded)

{0, 1, 2, 3, 4, 5, 6, 7}

Examine the test labels after encoding

set(test_labels_encoded)

{0, 1, 2, 3, 4, 5, 6, 7}

Make a custom Torch dataset

Here we combine the encoded labels and texts into dataset objects. We use the custom Torch
MyDataSet class to make a train_dataset object from the train_encodings and
train_labels_encoded. We also make a test_dataset object from test_encodings, and test_labels_encoded.

class MyDataset(torch.utils.data.Dataset):
    def __init__(self, encodings, labels):
        self.encodings = encodings
        self.labels = labels

    def __getitem__(self, idx):
        item = {key: torch.tensor(val[idx]) for key, val in self.encodings.items()}
        item['labels'] = torch.tensor(self.labels[idx])
        return item

    def __len__(self):
        return len(self.labels)

train_dataset = MyDataset(train_encodings, train_labels_encoded)
test_dataset = MyDataset(test_encodings, test_labels_encoded)

Examine a Goodreads review in the Torch training_dataset after encoding

' '.join(train_dataset.encodings[0].tokens[0:100])

Examine a Goodreads review in the Torch test_dataset after encoding

' '.join(test_dataset.encodings[1].tokens[0:100])

Load pre-trained BERT model

Here we load a pre-trained DistilBERT model and send it to CUDA.

Note: If you decide to repeat fine-tuning after already running the following cells, make sure that you re-run this cell to re-load the original pre-trained model before fine-tuning again.
Some weights of the model checkpoint at distilbert-base-cased were not used when
- This IS expected if you are initializing DistilBertForSequenceClassification f
- This IS NOT expected if you are initializing DistilBertForSequenceClassification

Some weights of DistilBertForSequenceClassification were not initialized from th
You should probably TRAIN this model on a down-stream task to be able to use it

Set the BERT fine-tuning parameters

These are the arguments we’ll set in the HuggingFace TrainingArguments objects, which we’ll then
pass to the HuggingFace Trainer object. There are many more possible arguments, but here we
highlight the basics and some common gotchas.

When training your own model, you should search over these parameters to find the best settings
for your particular dataset. You should use a held-out set of validation data for this step.

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Explanation</th>
</tr>
</thead>
<tbody>
<tr>
<td>num_train_epochs</td>
<td>total number of training epochs (how many times to pass through the entire dataset; too much can cau</td>
</tr>
<tr>
<td>per_device_train_batch_size</td>
<td>batch size per device during training</td>
</tr>
<tr>
<td>per_device_eval_batch_size</td>
<td>batch size for evaluation</td>
</tr>
<tr>
<td>warmup_steps</td>
<td>number of warmup steps for learning rate scheduler (set lower because of small dataset size)</td>
</tr>
<tr>
<td>weight_decay</td>
<td>strength of weight decay (reduces size of weights, like regularization)</td>
</tr>
<tr>
<td>output_dir</td>
<td>output directory for the fine-tuned model and configuration files</td>
</tr>
<tr>
<td>logging_dir</td>
<td>directory for storing logs</td>
</tr>
<tr>
<td>logging_steps</td>
<td>how often to print logging output (so that we can stop training early if the loss isn’t going down)</td>
</tr>
<tr>
<td>evaluation_strategy</td>
<td>evaluate while training so that we can see the accuracy going up</td>
</tr>
</tbody>
</table>

training_args = TrainingArguments(
    num_train_epochs=3,  # total number of training epochs
    per_device_train_batch_size=16,  # batch size per device during training
    per_device_eval_batch_size=20,  # batch size for evaluation
)
Fine-tune the BERT model

First, we define a custom evaluation function that returns the accuracy. You could modify this function to return precision, recall, F1, and/or other metrics.

```python
def compute_metrics(pred):
    labels = pred.label_ids
    preds = pred.predictions.argmax(-1)
    acc = accuracy_score(labels, preds)
    return {
        'accuracy': acc,
    }
```

Then we create a HuggingFace Trainer object using the TrainingArguments object that we created above. We also send our compute_metrics function to the Trainer object, along with our test and train datasets.

**Note:** This is what we've been aiming for this whole time! All the work of tokenizing, creating datasets, and setting the training arguments was for this cell.

```python
trainer = Trainer(
    model=model,  # the instantiated 😊 Transformers model to
    args=training_args,  # training arguments, defined above
    train_dataset=train_dataset,  # training dataset
    eval_dataset=test_dataset,  # evaluation dataset (usually a validation set)
    compute_metrics=compute_metrics  # our custom evaluation function
)
```

Time to finally fine-tune!
Be patient; if you've set everything in Colab to use GPUs, then it should only take a minute or two to run, but if you're running on CPU, it can take hours.

After every 10 steps (as we specified in the TrainingArguments object), the trainer will output the current state of the model, including the training loss, validation ("test") loss, and accuracy (from our compute_metrics function).

You should see the loss going down and the accuracy going up. If instead they are staying the

```python
trainer.train()
```

<table>
<thead>
<tr>
<th>Step</th>
<th>Training Loss</th>
<th>Validation Loss</th>
<th>Accuracy</th>
</tr>
</thead>
<tbody>
<tr>
<td>100</td>
<td>1.996600</td>
<td>1.705313</td>
<td>0.373125</td>
</tr>
<tr>
<td>200</td>
<td>1.584600</td>
<td>1.385085</td>
<td>0.513750</td>
</tr>
<tr>
<td>300</td>
<td>1.445400</td>
<td>1.391928</td>
<td>0.489375</td>
</tr>
<tr>
<td>400</td>
<td>1.386400</td>
<td>1.374399</td>
<td>0.500000</td>
</tr>
<tr>
<td>500</td>
<td>1.101900</td>
<td>1.345952</td>
<td>0.528125</td>
</tr>
<tr>
<td>600</td>
<td>1.086500</td>
<td>1.306563</td>
<td>0.540625</td>
</tr>
<tr>
<td>700</td>
<td>1.100300</td>
<td>1.269690</td>
<td>0.546875</td>
</tr>
<tr>
<td>800</td>
<td>1.102800</td>
<td>1.269377</td>
<td>0.532500</td>
</tr>
<tr>
<td>900</td>
<td>0.752600</td>
<td>1.316513</td>
<td>0.546875</td>
</tr>
<tr>
<td>1000</td>
<td>0.745900</td>
<td>1.337238</td>
<td>0.553125</td>
</tr>
<tr>
<td>1100</td>
<td>0.713400</td>
<td>1.345329</td>
<td>0.552500</td>
</tr>
<tr>
<td>1200</td>
<td>0.678900</td>
<td>1.348691</td>
<td>0.550000</td>
</tr>
</tbody>
</table>

TrainOutput(global_step=1200, training_loss=1.141270949045817, metrics={'train_r

**Save fine-tuned model**

The following cell will save the model and its configuration files to a directory in Colab. To preserve this model for future use, you should download the model to your computer.

```python
trainer.save_model(cached_model_directory_name)
```
(Optional) If you've already fine-tuned and saved the model, you can reload it using the following line. You don't have to run fine-tuning every time you want to evaluate.

```python
# trainer = DistilBertForSequenceClassification.from_pretrained(cached_model_directory)
```

### Evaluate fine-tuned model

The following function of the `Trainer` object will run the built-in evaluation, including our `compute_metrics` function.

```python
trainer.evaluate()
```

```
[80/80 00:28]
{'epoch': 3.0,
 'eval_accuracy': 0.55,
 'eval_loss': 1.3486912250518799,
 'eval_mem_cpu_alloc_delta': 0,
 'eval_mem_cpu_peaked_delta': 0,
 'eval_mem_gpu_alloc_delta': 0,
 'eval_mem_gpu_peaked_delta': 723764736,
 'eval_runtime': 28.7647,
 'eval_samples_per_second': 55.624}
```

But we might want to do more fine-grained analysis of the model, so we extract the predicted labels.

```python
predicted_results = trainer.predict(test_dataset)
```

```
[80/80 00:57]
predicted_results.predictions.shape

(1600, 8)
```

```python
predicted_labels = predicted_results.predictions.argmax(-1)  # Get the highest probability
predicted_labels = predicted_labels.flatten().tolist()  # Flatten the predictions
predicted_labels = [id2label[l] for l in predicted_labels]  # Convert from integers to labels

len(predicted_labels)
```

1600
print(classification_report(test_labels, 
    predicted_labels))

<table>
<thead>
<tr>
<th></th>
<th>precision</th>
<th>recall</th>
<th>f1-score</th>
<th>support</th>
</tr>
</thead>
<tbody>
<tr>
<td>children</td>
<td>0.64</td>
<td>0.65</td>
<td>0.65</td>
<td>200</td>
</tr>
<tr>
<td>comics_graphic</td>
<td>0.60</td>
<td>0.72</td>
<td>0.66</td>
<td>200</td>
</tr>
<tr>
<td>fantasy_paranormal</td>
<td>0.36</td>
<td>0.32</td>
<td>0.34</td>
<td>200</td>
</tr>
<tr>
<td>history_biography</td>
<td>0.63</td>
<td>0.53</td>
<td>0.57</td>
<td>200</td>
</tr>
<tr>
<td>mystery_thriller_crime</td>
<td>0.52</td>
<td>0.60</td>
<td>0.56</td>
<td>200</td>
</tr>
<tr>
<td>poetry</td>
<td>0.74</td>
<td>0.80</td>
<td>0.77</td>
<td>200</td>
</tr>
<tr>
<td>romance</td>
<td>0.49</td>
<td>0.41</td>
<td>0.45</td>
<td>200</td>
</tr>
<tr>
<td>young_adult</td>
<td>0.37</td>
<td>0.37</td>
<td>0.37</td>
<td>200</td>
</tr>
<tr>
<td>accuracy</td>
<td></td>
<td></td>
<td>0.55</td>
<td>1600</td>
</tr>
<tr>
<td>macro avg</td>
<td>0.55</td>
<td>0.55</td>
<td>0.55</td>
<td>1600</td>
</tr>
<tr>
<td>weighted avg</td>
<td>0.55</td>
<td>0.55</td>
<td>0.55</td>
<td>1600</td>
</tr>
</tbody>
</table>

Pull out correct and incorrect classifications for examination

Let's use our predicted labels for some analysis!

Now that we've fine-tuned and pulled out our predicted labels, the BERT part of this tutorial is done. You can now use the predicted labels in the same way you would use any set of predicted labels from any classification model. We'll show some examples here.

First, let's print out some example predictions that were correct.

```python
for _true_label, _predicted_label, _text in random.sample(list(zip(test_labels, predicted_labels)), k):
    if _true_label == _predicted_label:
        print('LABEL:', _true_label)
        print('REVIEW TEXT:', _text[:100], '...')
        print()
```

LABEL: children
REVIEW TEXT: Only David Shannon can write a fun book about lice!
    Informative and sure to make us all itchy. ...

LABEL: poetry
REVIEW TEXT: Read as 3.5 stars...
    I found this collection to be excellent - intense, beautiful, sad, uncomforta .

LABEL: children
REVIEW TEXT: Animals, books, encouragement to read. Positively captivating. Plus
LABEL: romance  
REVIEW TEXT: loved this just as much as the others ...

LABEL: romance  
REVIEW TEXT: OMG!!! This book was awesome, but I have yet to be disappointed by

LABEL: history_biography  
REVIEW TEXT: A good look at Elizabeth's early life. Weir does so much research t

LABEL: poetry  
REVIEW TEXT: Exquisite! Favorite new poetry find of the year. ...

LABEL: poetry  
REVIEW TEXT: This book was created as a teaching tool. It includes what are supp

Now let's print out some misclassifications.

for _true_label, _predicted_label, _text in random.sample(list(zip(test_labels, predi
if _true_label != _predicted_label:
    print('TRUE LABEL:', _true_label)
    print('PREDICTED LABEL:', _predicted_label)
    print('REVIEW TEXT:', _text[:100], '...')
    print()

TRUE LABEL: poetry  
PREDICTED LABEL: comics_graphic  
REVIEW TEXT: There are tears. ...

TRUE LABEL: mystery_thriller_crime  
PREDICTED LABEL: romance  
REVIEW TEXT: I feel like I've found a pot of gold at the end of the rainbow with

TRUE LABEL: comics_graphic  
PREDICTED LABEL: children  
REVIEW TEXT: I'm talking about these books by Jeffrey Brown.  
    The books have brightly colored pictures that are ...

TRUE LABEL: young_adult  
PREDICTED LABEL: romance  
REVIEW TEXT: This unique story is told from the alternating views of Emma and Be

TRUE LABEL: romance  
PREDICTED LABEL: young_adult  
REVIEW TEXT: THE STORY  
    In a country divide by castes it's every girl dream to be a Selected and have t

TRUE LABEL: young_adult  
PREDICTED LABEL: mystery_thriller_crime  
REVIEW TEXT: one of those unforgettable books. ...

TRUE LABEL: mystery_thriller_crime  
PREDICTED LABEL: history_biography
Although it could have done a better job of going into detail about ...(TRUE LABEL: comics_graphic, PREDICTED LABEL: fantasy_paranormal)

3,5 ... (TRUE LABEL: history_biography, PREDICTED LABEL: comics_graphic)

I think she is...full of crap. My social worker friend and her coll ...(TRUE LABEL: romance, PREDICTED LABEL: young_adult)

OMG I stayed up until 4 in the morning to finish this book!!!! I lc ...(TRUE LABEL: young_adult, PREDICTED LABEL: mystery_thriller_crime)

Es dauerte ein Weilchen, bis ich mich in die Geschichte einfinden konnte. Anfan ...(TRUE LABEL: history_biography, PREDICTED LABEL: mystery_thriller_crime)

I'll admit it: I was hesitant to start this book. I was so sad afte ...(TRUE LABEL: history_biography, PREDICTED LABEL: mystery_thriller_crime)

Finally, let's create some heatmaps to examine misclassification patterns. We could use these patterns to think about similarities and differences between genres, according to book reviewers.

genre_classifications_dict = defaultdict(int)
for _true_label, _predicted_label in zip(test_labels, predicted_labels):
    genre_classifications_dict[(_true_label, _predicted_label)] += 1

dicts_to_plot = []
for (_true_genre, _predicted_genre), _count in genre_classifications_dict.items():
    dicts_to_plot.append({'True Genre': _true_genre,
                          'Predicted Genre': _predicted_genre,
                          'Number of Classifications': _count})

df_to_plot = pd.DataFrame(dicts_to_plot)
df_wide = df_to_plot.pivot_table(index='True Genre',
columns='Predicted Genre',
values='Number of Classifications')

plt.figure(figsize=(9,7))
sns.set(style='ticks', font_scale=1.2)
sns.heatmap(df_wide, linewidths=1, cmap='Purples')
plt.xticks(rotation=45, ha='right')
plt.tight_layout()
plt.show()
Looks good! We can see that overall, our model is assigning the correct labels for each genre.

Now, let's remove the diagonal from the plot to highlight the misclassifications.

genre_classifications_dict = defaultdict(int)
for _true_label, _predicted_label in zip(test_labels, predicted_labels):
    if _true_label != _predicted_label:  # Remove the diagonal to highlight misclassifications
        genre_classifications_dict[(_true_label, _predicted_label)] += 1

dicts_to_plot = []
for (_true_genre, _predicted_genre), _count in genre_classifications_dict.items():
    dicts_to_plot.append({
        'True Genre': _true_genre,
        'Predicted Genre': _predicted_genre,
        'Number of Classifications': _count})

df_to_plot = pd.DataFrame(dicts_to_plot)
df_wide = df_to_plot.pivot_table(index='True Genre',
                                   columns='Predicted Genre',
                                   values='Number of Classifications')

plt.figure(figsize=(9, 7))
sns.set(style='ticks', font_scale=1.2)
sns.heatmap(df_wide, linewidths=1, cmap='Purples')
plt.xticks(rotation=45, ha='right')
There's much more you can do with your own dataset and labels! Classification can be used to apply a small set of labels across a big dataset; to explore misclassifications to better understand users; and much more! We hope you'll use this tutorial in all kinds of creative ways.